

CS 31: Intro to Systems

Arrays, Structs, Strings, and Pointers

Kevin Webb

Swarthmore College

October 20, 2022

Reading Quiz

Overview

- Accessing *things* via an offset
 - Arrays, Structs, Unions
- How complex structures are stored in memory
 - Multi-dimensional arrays & Structs

So far: Primitive Data Types

- We've been using ints, floats, chars, pointers
- Simple to place these in memory:
 - They have an unambiguous size
 - They fit inside a register*
 - The hardware can operate on them directly

(*There are special registers for floats and doubles that use the IEEE floating point format.)

Composite Data Types

- Combination of one or more existing types into a new type. (e.g., an array of *multiple* ints, or a struct)
- Example: a queue
 - Might need a value (int) plus a link to the next item (pointer)

```
struct queue_node{  
    int value;  
    struct queue_node *next;  
}
```


Base + Offset

- We know that arrays act as a pointer to the first element. For bucket [N], we just skip forward N.



- “We’re goofy computer scientists who count starting from zero.”

Base + Offset

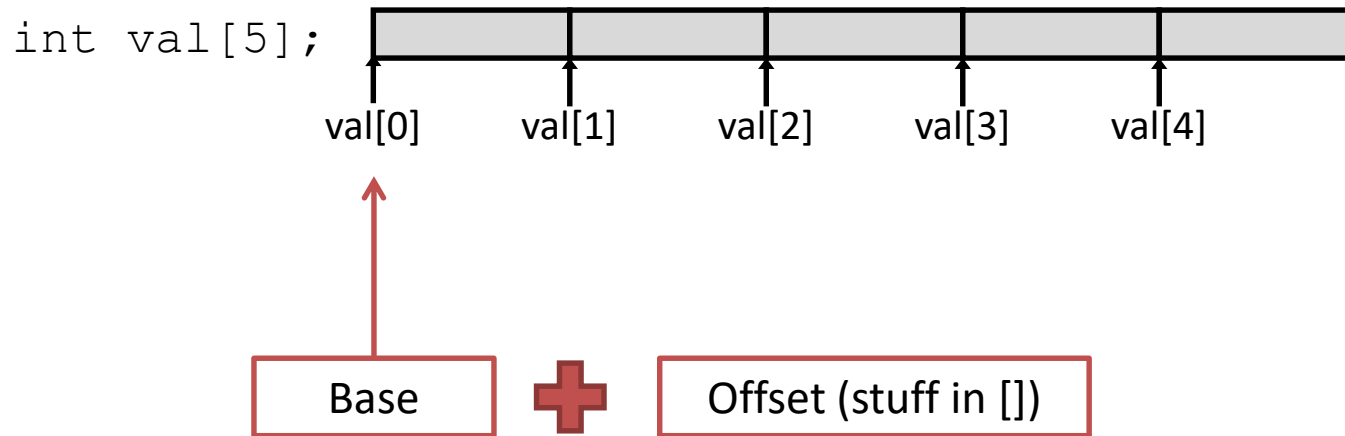
- We know that arrays act as a pointer to the first element. For bucket [N], we just skip forward N.



- ~~• “We’re goofy computer scientists who count starting from zero.”~~

Base + Offset

- We know that arrays act as a pointer to the first element. For bucket [N], we just skip forward N.



This is why we start counting from zero!
Skipping forward with an offset of zero (`[0]`) gives us the first bucket...

Which expression would compute the address of `iptr[3]`?

↑
What if this isn't known at compile time?

- A. $0x0824 + 3 * 4$
- B. $0x0824 + 4 * 4$
- C. $0x0824 + 0xC$
- D. More than one (which?)
- E. None of these

Heap	
0x0824:	<code>iptr[0]</code>
0x0828:	<code>iptr[1]</code>
0x082C:	<code>iptr[2]</code>
0x0830:	<code>iptr[3]</code>

Recall: Addressing Mode: Memory

2. access address in register + immediate: [register, #constant]

3. access address in register + register: [register, register]

- Examples:

- `ldr x3, [sp, #8]` (Take the value in `sp` (`x31`), add 8 to it, and treat the sum as a memory address. Load the data found at that memory address into register `x3`.)

- `str x2, [x1, x7]` (Take the value in `x1`, add the value stored in `x7` to it, and treat the sum as a memory address. Load the data found at that memory address into register `x2`.)

Recall: Addressing Mode: Memory

2. access address in register + immediate: [register, #constant]
 - Helpful for accessing memory relative to stack pointer
3. access address in register + register: [register, register]

Recall: Addressing Mode: Memory

2. access address in register + immediate: [register, #constant]
 - Helpful for accessing memory relative to stack pointer
 - Helpful for accessing a field of a struct

3. access address in register + register: [register, register]
 - Helpful for accessing an element of an array

Recall: Addressing Mode: Memory

2. access address in register + immediate: [register, #constant]
 - Helpful for accessing memory relative to stack pointer
 - Helpful for accessing a field of a struct

3. access address in register + register: [register, register]
 - **Helpful for accessing an element of an array**

Example

Suppose:

iptr is stored in register x9.

i is at sp+8, and equals 2.

User says:

```
iptr[i] = 9;
```

Translates to:

x9: Array base address



Registers:

x9	0x0824
x10	
x11	9

Heap			
0x0824:	iptr[0]		
0x0828:	iptr[1]		
0x082C:	iptr[2]		
0x0830:	iptr[3]		

Example

Suppose:

iptr is stored in register x9.

i is at sp+8, and equals 2.

User says:

```
iptr[i] = 9;
```

Translates to:

```
ldr x10, [sp, #8]
```

x9: Array base address



Registers:

x9	0x0824
x10	2
x11	9

Heap			
0x0824:	iptr[0]		
0x0828:	iptr[1]		
0x082C:	iptr[2]		
0x0830:	iptr[3]		

Example

Suppose:

iptr is stored in register x9.

i is at sp+8, and equals 2.

User says:

```
iptr[i] = 9;
```

Translates to:

```
ldr x10, [sp, #8]
```

```
lsl x10, x10, #2
```

Multiply index by 4, the size of an int.

x9: Array base address

Registers:

x9	0x0824
x10	8
x11	9

Heap			
0x0824:	iptr[0]		
0x0828:	iptr[1]		
0x082C:	iptr[2]		
0x0830:	iptr[3]		

Example

Suppose:

iptr is stored in register x9.

i is at sp+8, and equals 2.

User says:

```
iptr[i] = 9;
```

Translates to:

```
ldr x10, [sp, #8]
lsl x10, x10, #2
str x11, [x9, x10]
```

x9: Array base address

Registers:

x9	0x0824
x10	8
x11	9

Heap			
0x0824:	iptr[0]		
0x0828:	iptr[1]		
0x082C:	iptr[2]		
0x0830:	iptr[3]		

Example

Suppose:

iptr is stored in register x9.

i is at sp+8, and equals 2.

User says:

```
iptr[i] = 9;
```

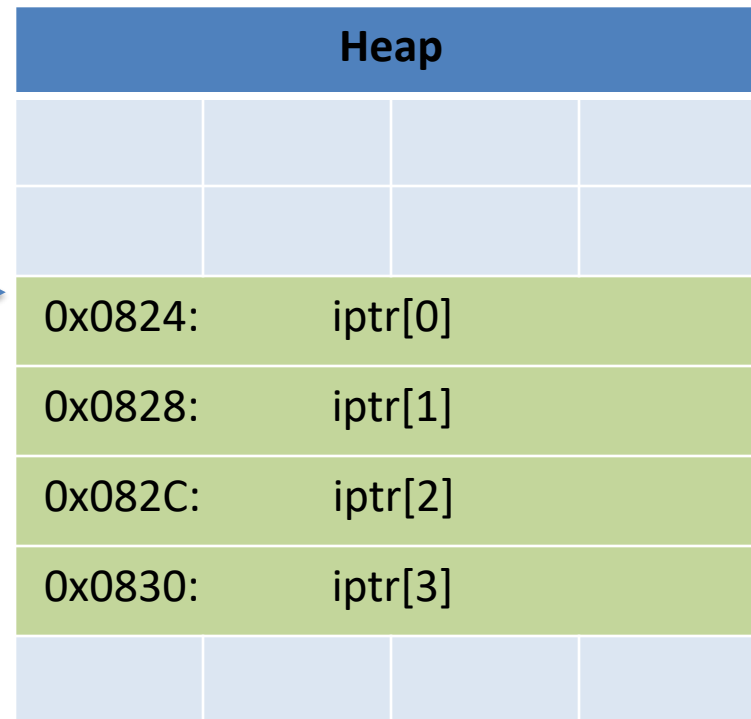
Translates to:

```
ldr x10, [sp, #8]
lsl x10, x10, #2
str x11, [x9, x10]
```

x9: Array base address

Registers:

x9	0x0824
x10	8
x11	9



Example

Suppose:

iptr is stored in register x9.

i is at sp+8, and equals 2.

User says:

```
iptr[i] = 9;
```

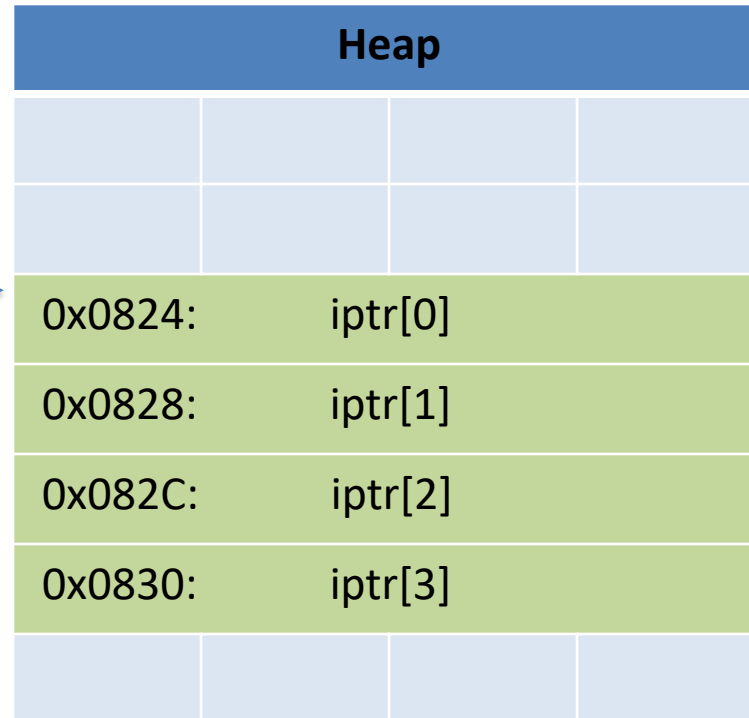
Translates to:

```
ldr x10, [sp, #8]
lsl x10, x10, #2
str x11, [x9, x10]
```

x9: Array base address

Registers:

x9	0x0824
x10	8
x11	9



From here, if the program increments i (e.g., in a loop) and accesses the array at the new (incremented) position of i :

Compiler can simply add 4 (size of an int) to $x10$ and continue to load / store at: $[x9, x10]$

Translates to:

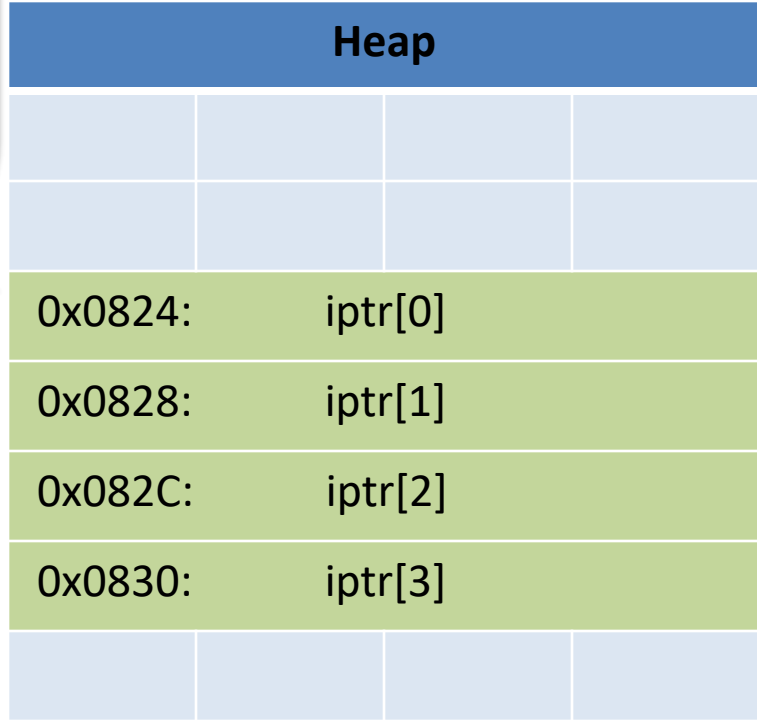
```
ldr x10, [sp, #8]
lsl x10, x10, #2
str x11, [x9, x10]
```

e

x9: Array base address

Registers:

x9	0x0824
x10	8
x11	9



Recall: Addressing Mode: Memory

2. access address in register + immediate: [register, #constant]
 - Helpful for accessing memory relative to stack pointer
 - **Helpful for accessing a field of a struct**

3. access address in register + register: [register, register]
 - Helpful for accessing an element of an array

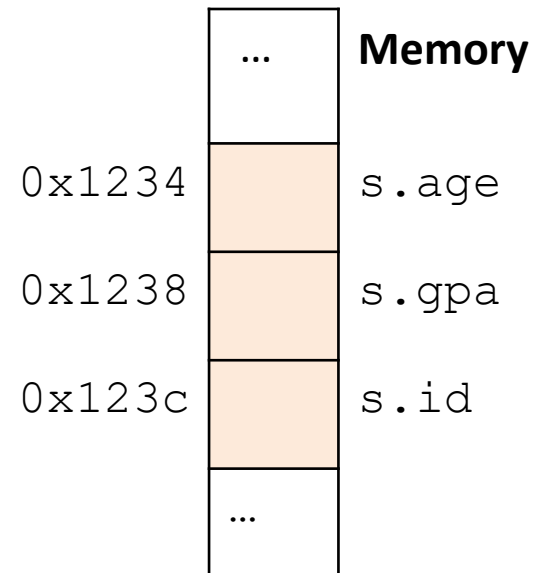
Structs

- Multiple values (fields) stored together
 - Defines a new type in C's type system
- Laid out contiguously by field (with a caveat we'll see later)
 - In order of field declaration.

Structs

- Laid out contiguously by field (with a caveat we'll see later)
 - In order of field declaration.

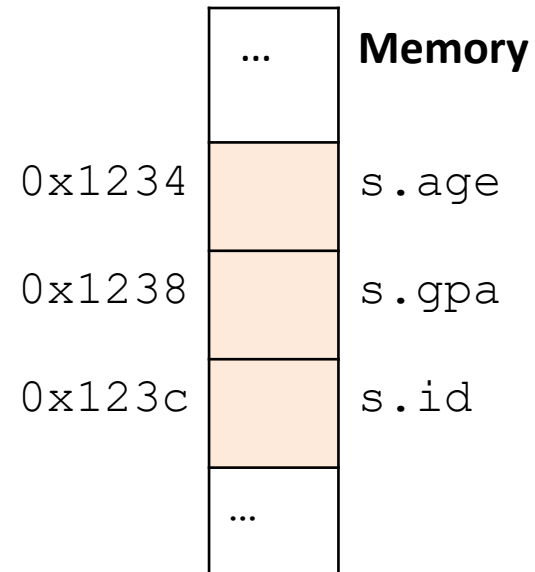
```
struct student{  
    int age;  
    float gpa;  
    int id;  
};  
  
struct student s;
```



Structs

- Struct fields accessible as a base + displacement
 - Compiler knows (constant) displacement of each field

```
struct student{  
    int age;  
    float gpa;  
    int id;  
};  
  
struct student s;
```

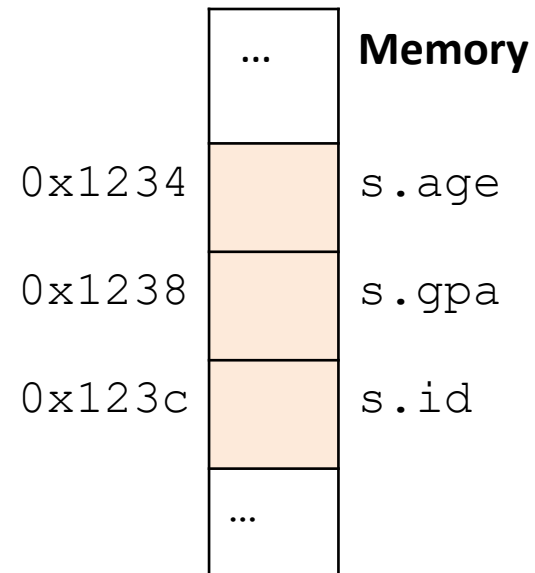


Structs

- Struct fields accessible as a base + displacement
 - Compiler knows (constant) displacement of each field

```
struct student{  
    int age;  
    float gpa;  
    int id;  
};
```

```
struct student s;  
s.id = 12;
```



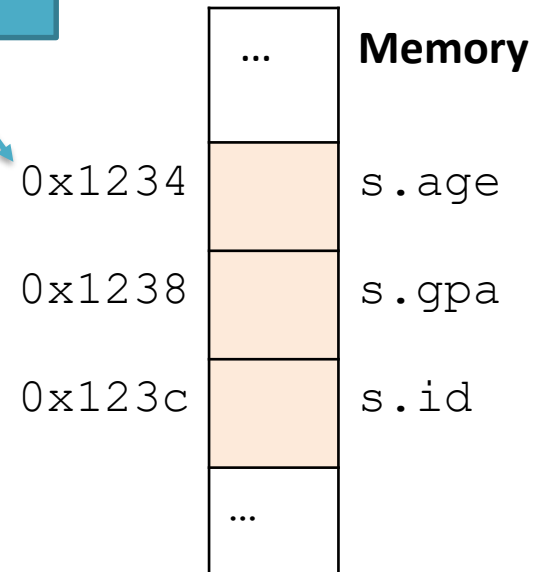
Structs

- Struct fields accessible as a base + displacement
 - Compiler knows (constant) displacement of each field

```
struct student{  
    int age;  
    float gpa;  
    int id;  
};
```

```
struct student s;  
s.id = 12;
```

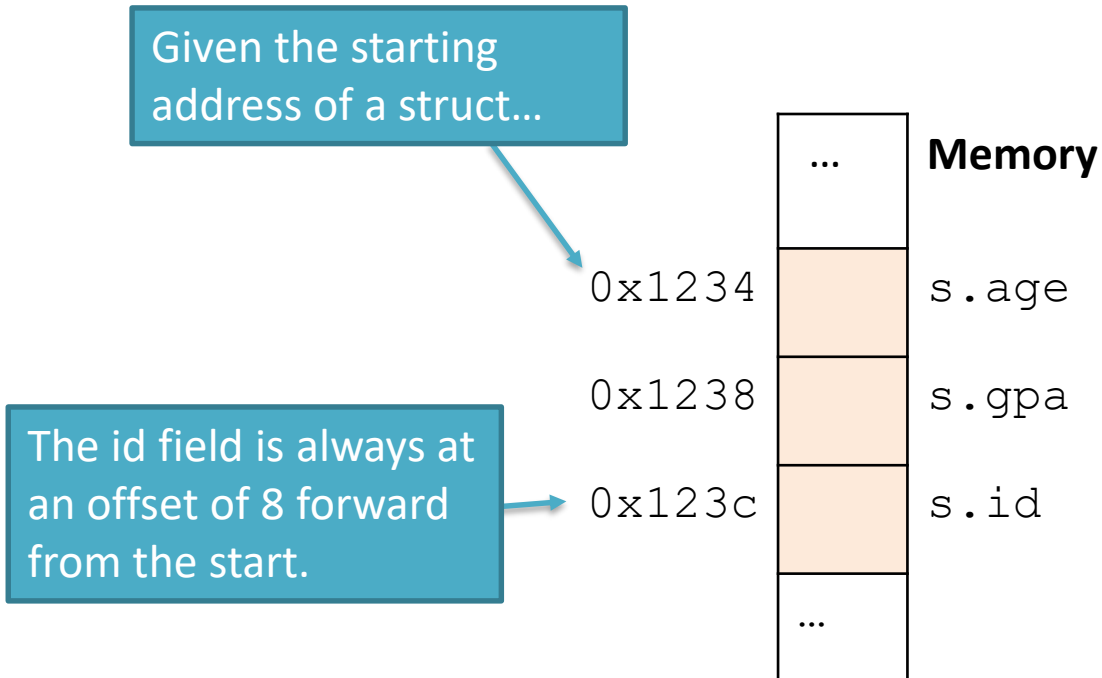
Given the starting
address of a struct...



Structs

- Struct fields accessible as a base + displacement
 - Compiler knows (constant) displacement of each field

```
struct student{  
    int age;  
    float gpa;  
    int id;  
};  
  
struct student s;  
s.id = 12;
```



Structs

- Struct fields accessible as a base + displacement
 - Compiler knows (constant) displacement of each field

In assembly:

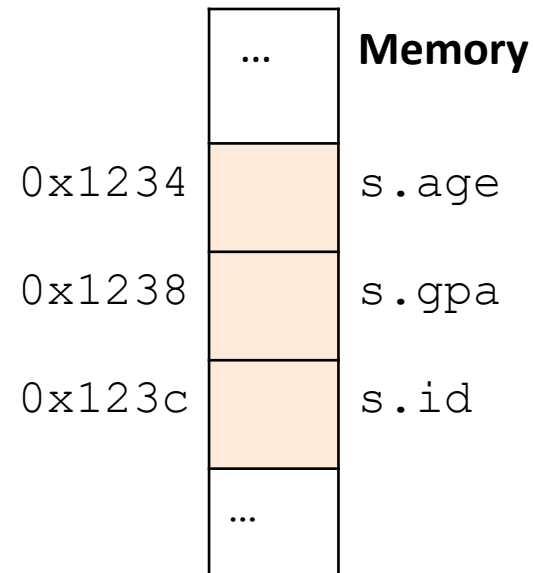
```
str reg_value, [reg_base, #8]
```

Where:

reg_value is a register holding the value to store (12)

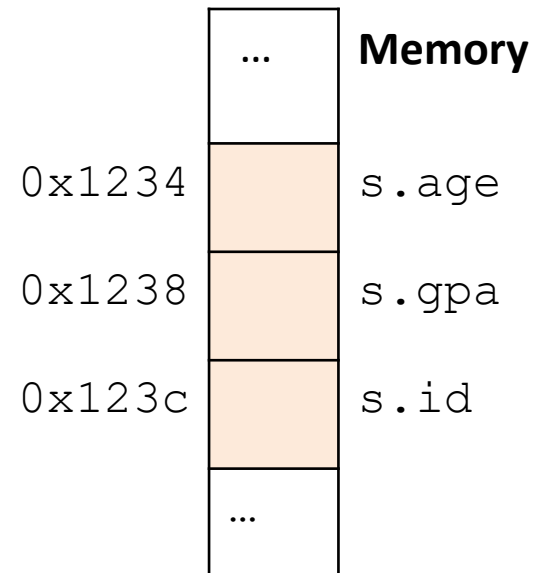
reg_base is a register holding the base address of the struct

```
s.id = 12;
```



Structs

- Laid out contiguously by field
 - In order of field declaration.
 - May require some padding, for alignment.



Data Alignment:

- Where (which address) can a field be located?
- char (1 byte): can be allocated at any address:
0x1230, 0x1231, 0x1232, 0x1233, 0x1234, ...
- short (2 bytes): must be aligned on 2-byte addresses:
0x1230, 0x1232, 0x1234, 0x1236, 0x1238, ...
- int (4 bytes): must be aligned on 4-byte addresses:
0x1230, 0x1234, 0x1238, 0x123c, 0x1240, ...

Why do we want to align data on multiples of the data size?

- A. It makes the hardware faster.
- B. It makes the hardware simpler.
- C. It makes more efficient use of memory space.
- D. It makes implementing the OS easier.
- E. Some other reason.

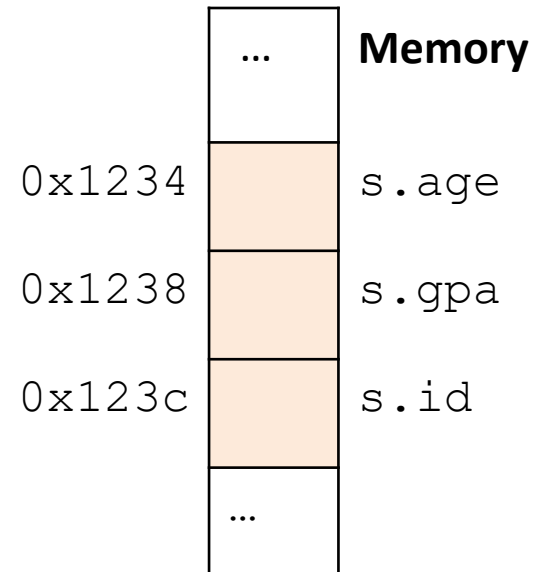
Data Alignment: Why?

- Simplify hardware
 - e.g., only read ints from multiples of 4
 - Don't need to build wiring to access 4-byte chunks at any arbitrary location in hardware
- Inefficient to load/store single value across alignment boundary (1 vs. 2 loads)
- Simplify OS:
 - Prevents data from spanning virtual pages
 - Atomicity issues with load/store across boundary

Structs

- Laid out contiguously by field
 - In order of field declaration.
 - May require some padding, for alignment.

```
struct student{  
    int age;  
    float gpa;  
    int id;  
};  
  
struct student s;
```



Structs

```
struct student{  
    char name[11];  
    short age;  
    int id;  
};
```

How much space do we need to store one of these structures?

```
struct student{  
    char name[11];  
    short age;  
    int id;  
};
```

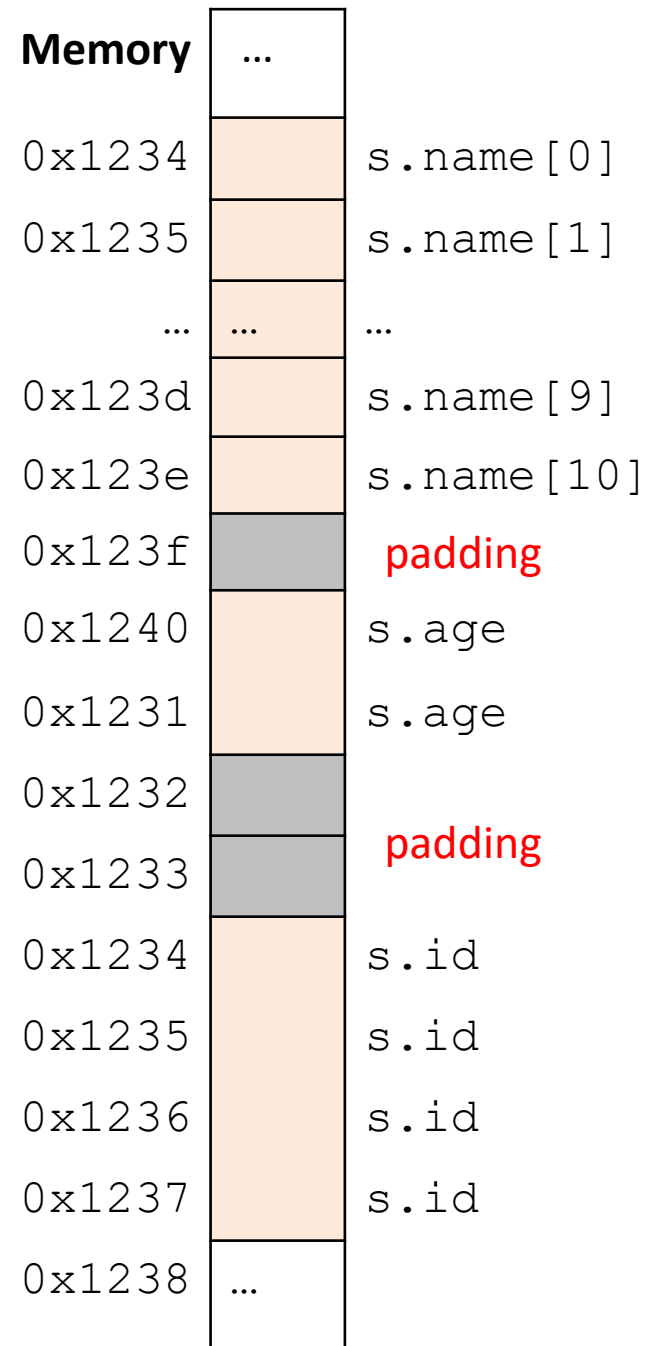
- A. 17 bytes
- B. 18 bytes
- C. 20 bytes
- D. 22 bytes
- E. 24 bytes

Structs

```
struct student{  
    char name[11];  
    short age;  
    int id;  
};
```

- Size of data: 17 bytes
- Size of struct: 20 bytes

Use sizeof() when allocating structs with malloc()!



Alternative Layout

```
struct student{  
    int id;  
    short age;  
    char name[11];  
};
```



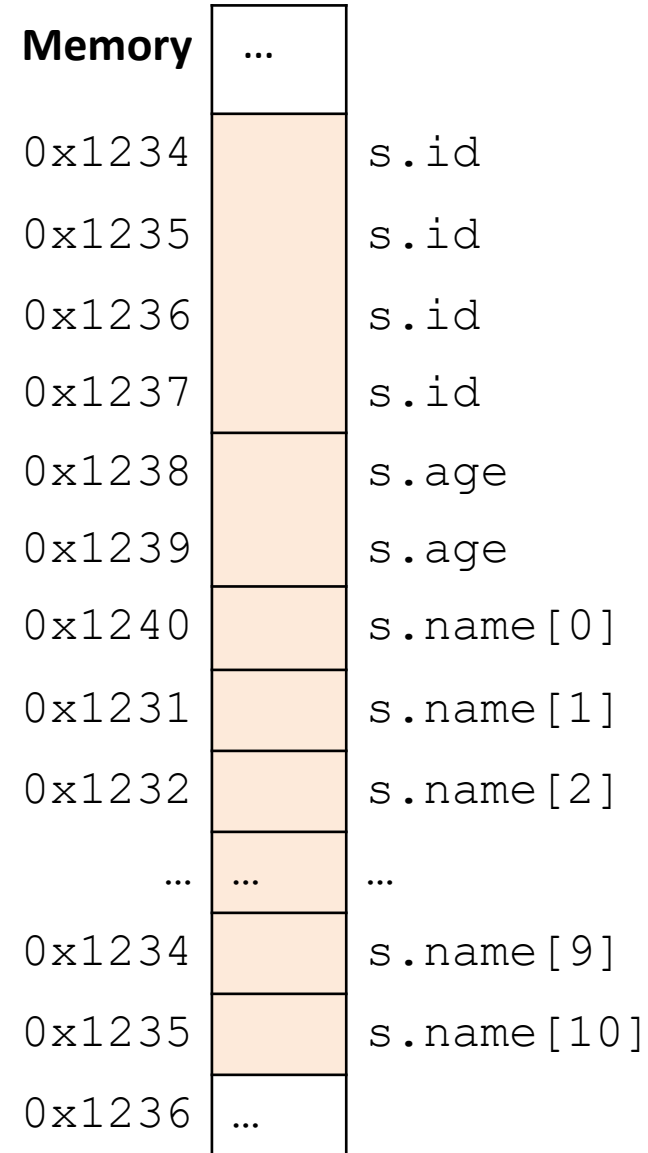
Same fields, declared in
a different order.

Alternative Layout

```
struct student{  
    int id;  
    short age;  
    char name[11];  
};
```

- Size of data: 17 bytes
- Size of struct: 17 bytes!

In general, this isn't a big deal on a day-to-day basis. Don't go out and rearrange all your struct declarations.



Aside: Network Headers

- In networks, we attach metadata to packets
 - Things like destination address, port #, etc.
- Common for these to be a specific size/format
 - e.g., the first 20 bytes must be laid out like ...
- Naïvely declaring a struct might introduce padding, violate format.

Cool, so we can get rid of this struct padding by being smart about declarations?

A. Yes (why?)

B. No (why not?)

Cool, so we can get rid of this padding by being smart about declarations?

- Answer: Maybe.
- Rearranging helps, but often padding after the struct can't be eliminated.

```
struct T1 {  
    char c1;  
    char c2;  
    int x;  
};
```

```
struct T2 {  
    int x;  
    char c1;  
    char c2;  
};
```

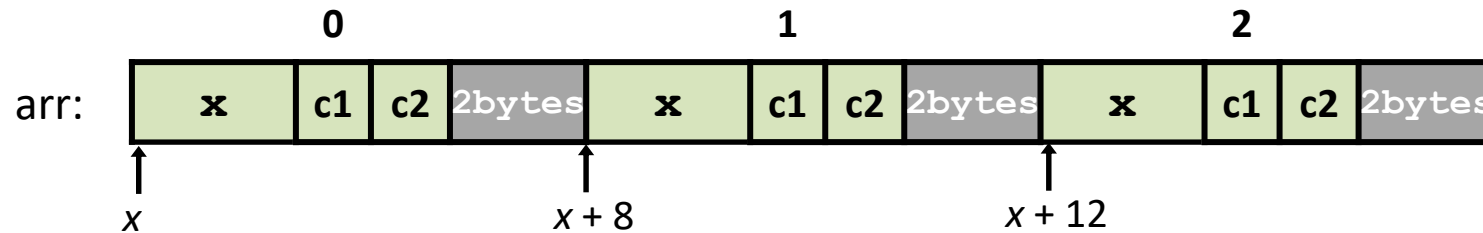


“External” Padding

- Array of Structs

Field values in each bucket must be properly aligned:

```
struct T2 arr[3];
```



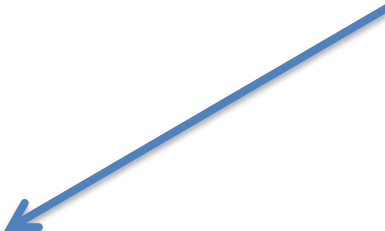
Buckets must be on a 4-byte aligned address

Struct field syntax...

```
struct student {  
    int id;  
    short age;  
    char name[11];  
};  
struct student s;
```

```
s.id = 406432;  
s.age = 20;  
strcpy(s.name, "Alice");
```

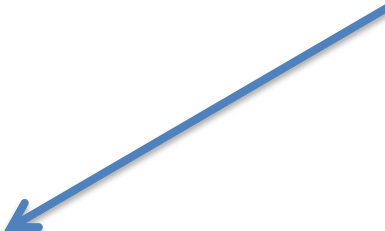
Struct is declared on
the stack.
(NOT a pointer)



Struct field syntax...

```
struct student {  
    int id;  
    short age;  
    char name[11];  
};
```

Not a struct, but a
pointer to a struct!



```
struct student *s = malloc(sizeof(struct student));
```

```
(*s).id = 406432;  
(*s).age = 20;  
strcpy((*s).name, "Alice");
```

This works, but is very ugly.

```
s->id = 406432;  
s->age = 20;  
strcpy(s->name, "Alice");
```

Access the struct field from a pointer with ->
Does a dereference and gets the field.

Stack Padding

- Memory alignment applies elsewhere too.

```
int x;           vs.      double y;  
char ch[5];     int x;  
short s;        short s;  
double y;       char ch[5];
```

In near all cases, you shouldn't stress about this. The compiler will figure out where to put things.

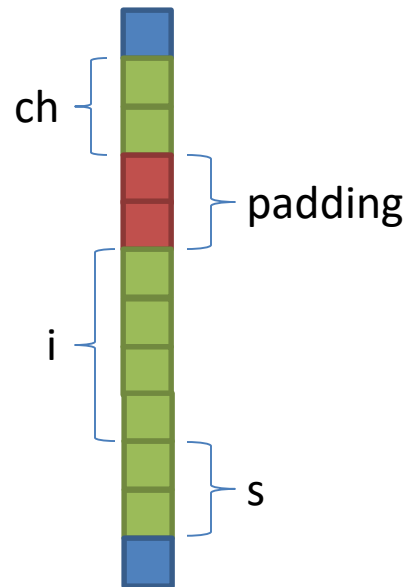
Exception: you're writing an OS and/or are optimizing for caches.

Unions

- Declared like a struct, but only contains one field, rather than all of them.
- Struct: field 1 and field 2 and field 3 ...
- Union: field 1 or field 2 or field 3 ...
- Intuition: you know you only need to store one of N things, don't waste space.

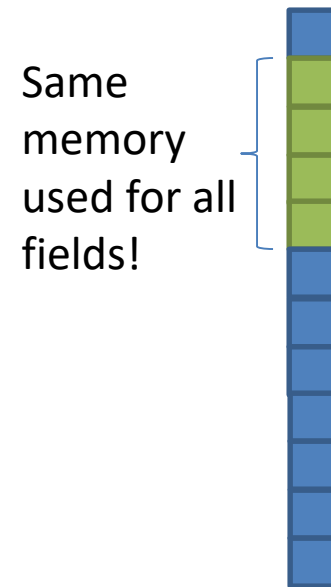
Unions

```
struct my_struct {  
    char ch[2];  
    int i;  
    short s;  
}
```



my_struct in memory

```
union my_union {  
    char ch[2];  
    int i;  
    short s;  
}
```



my_union in memory

Unions

```
my_union u;
```

```
u.i = 7;
```

```
union my_union {
```

```
    char ch[2];
```

```
    int i;
```

```
    short s;
```

```
}
```

Same
memory
used for all
fields!



my_union in memory

Unions

```
my_union u;
```

```
u.i = 7;
```

```
u.s = 2;
```

```
union my_union {
```

```
    char ch[2];
```

```
    int i;
```

```
    short s;
```

```
}
```

Same
memory
used for all
fields!



my_union in memory

Unions

```
my_union u;
```

```
u.i = 7;
```

```
u.s = 2;
```

```
u.ch[0] = 'a';
```

Reading i or s here would be bad!

```
union my_union {  
    char ch[2];  
    int i;  
    short s;  
}
```

Same
memory
used for all
fields!



my_union in memory

Unions

```
my_union u;
```

```
u.i = 7;
```

```
u.s = 2;
```

```
u.ch[0] = 'a';
```

Reading i or s here would be bad!

```
u.i = 5;
```

```
union my_union {  
    char ch[2];  
    int i;  
    short s;  
}
```

Same
memory
used for all
fields!



my_union in memory

Unions

- You probably won't use these often.
- Use when you need mutually exclusive types.
- Can save memory.

```
union my_union {  
    char ch[2];  
    int i;  
    short s;  
}
```

Same
memory
used for all
fields!



my_union in memory

Two-dimensional Arrays

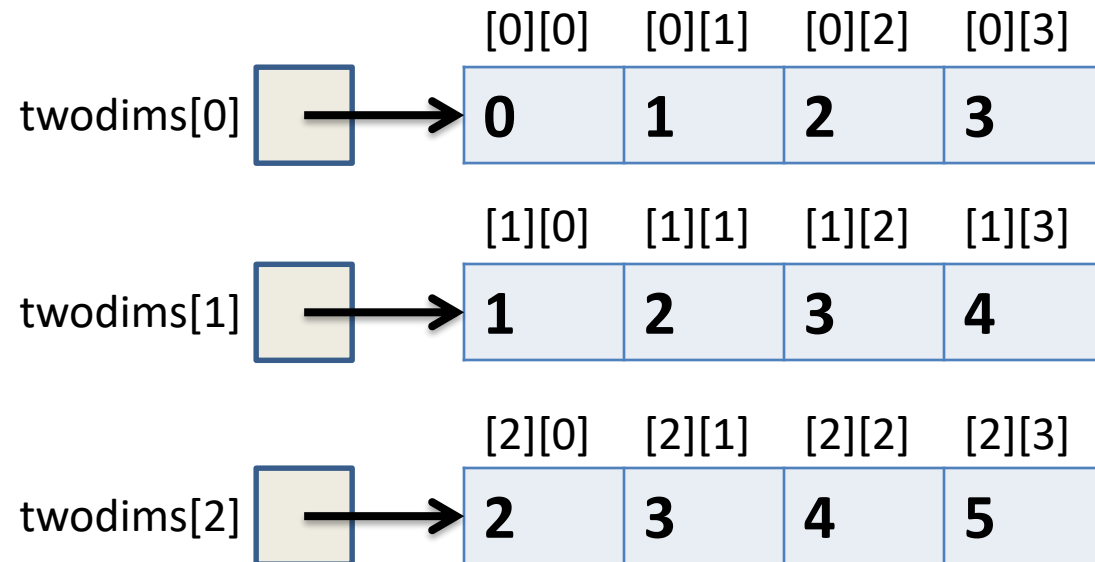
- Why stop at an array of ints?
How about an array of arrays of ints?

```
int twodims[3][4];
```

- “Give me three sets of four integers.”
- How should these be organized in memory?

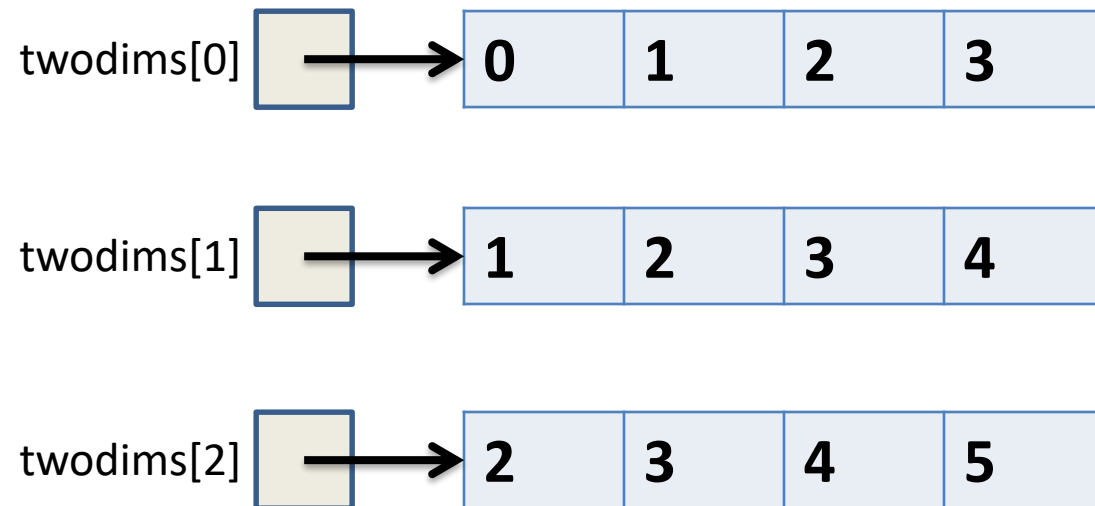
Two-dimensional Arrays

```
int twodims[3][4];  
for(i=0; i<3; i++) {  
    for(j=0; j<4; j++) {  
        twodims[i][j] = i+j;  
    }  
}
```



Two-dimensional Arrays: Matrix

```
int twodims[3][4];  
for(i=0; i<3; i++) {  
    for(j=0; j<4; j++) {  
        twodims[i][j] = i+j;  
    }  
}
```



Memory Layout

- Matrix: 3 rows, 4 columns

0	1	2	3
1	2	3	4
2	3	4	5

Row Major Order:

all Row 0 buckets, followed by
all Row 1 buckets, followed by
all Row 2 buckets, ...

0xf260	0	twodim[0][0]
0xf264	1	twodim[0][1]
0xf268	2	twodim[0][2]
0xf26c	3	twodim[0][3]
0xf270	1	twodim[1][0]
0xf274	2	twodim[1][1]
0xf278	3	twodim[1][2]
0xf27c	4	twodim[1][3]
0xf280	2	twodim[2][0]
0xf284	3	twodim[2][1]
0xf288	4	twodim[2][2]
0xf28c	5	twodim[2][3]

Memory Layout

- Matrix: 3 rows, 4 columns

0	1	2	3
1	2	3	4
2	3	4	5

`twodim[1][3]:`

base addr + row offset + col offset

`twodim + 1*ROWSIZE*4 + 3*4`

`0xf260 + 16 + 12 = 0xf27c`

<code>0xf260</code>	0	<code>twodim[0][0]</code>
<code>0xf264</code>	1	<code>twodim[0][1]</code>
<code>0xf268</code>	2	<code>twodim[0][2]</code>
<code>0xf26c</code>	3	<code>twodim[0][3]</code>
<code>0xf270</code>	1	<code>twodim[1][0]</code>
<code>0xf274</code>	2	<code>twodim[1][1]</code>
<code>0xf278</code>	3	<code>twodim[1][2]</code>
<code>0xf27c</code>	4	<code>twodim[1][3]</code>
<code>0xf280</code>	2	<code>twodim[2][0]</code>
<code>0xf284</code>	3	<code>twodim[2][1]</code>
<code>0xf288</code>	4	<code>twodim[2][2]</code>
<code>0xf28c</code>	5	<code>twodim[2][3]</code>

If we declared `int matrix[5][3];`,
and the base of matrix is `0x3420`, what is
the address of `matrix[3][2]`?

- A. `0x3438`
- B. `0x3440`
- C. `0x3444`
- D. `0x344C`
- E. None of these

Dynamic Two-dimensional Array

- Given the *row-major order* layout, a "two-dimensional array" is still just a contiguous block of memory:
- The malloc function returns... a pointer to a contiguous block of memory!

0xf260	0	twodim[0][0]
0xf264	1	twodim[0][1]
0xf268	2	twodim[0][2]
0xf26c	3	twodim[0][3]
0xf270	1	twodim[1][0]
0xf274	2	twodim[1][1]
0xf278	3	twodim[1][2]
0xf27c	4	twodim[1][3]
0xf280	2	twodim[2][0]
0xf284	3	twodim[2][1]
0xf288	4	twodim[2][2]
0xf28c	5	twodim[2][3]

Dynamic Two-dimensional Array

- For this example with three rows and four columns:

0	1	2	3
1	2	3	4
2	3	4	5

```
int *matrix = malloc(3 * 4 * sizeof(int));
```

Caveat: the C compiler doesn't know that you're planning to use this block of memory with more than one index (i.e., row and column).

Can't access: `matrix[i][j]`

0xf260	0	matrix[?]
0xf264	1	matrix[?]
0xf268	2	matrix[?]
0xf26c	3	matrix[?]
0xf270	1	matrix[?]
0xf274	2	matrix[?]
0xf278	3	matrix[?]
0xf27c	4	matrix[?]
0xf280	2	matrix[?]
0xf284	3	matrix[?]
0xf288	4	matrix[?]
0xf28c	5	matrix[?]

Dynamic Two-dimensional Array

- For this example with three rows and four columns:

0	1	2	3
1	2	3	4
2	3	4	5

```
int *matrix = malloc(3 * 4 * sizeof(int));
```

```
// Compute the offset manually
```

```
index = i * ROWSIZE + j;
```

```
matrix[index] = ...
```

0xf260	0	matrix[0 + 0]
0xf264	1	matrix[0 + 1]
0xf268	2	matrix[0 + 2]
0xf26c	3	matrix[0 + 3]
0xf270	1	matrix[4 + 0]
0xf274	2	matrix[4 + 1]
0xf278	3	matrix[4 + 2]
0xf27c	4	matrix[4 + 3]
0xf280	2	matrix[8 + 0]
0xf284	3	matrix[8 + 1]
0xf288	4	matrix[8 + 2]
0xf28c	5	matrix[8 + 3]

Two-dimensional array alternative

- (Dynamically) Allocate an array of pointers.
For each pointer, (dynamically) allocate an array.
- How do we get an array of pointers?

Two-dimensional array alternative

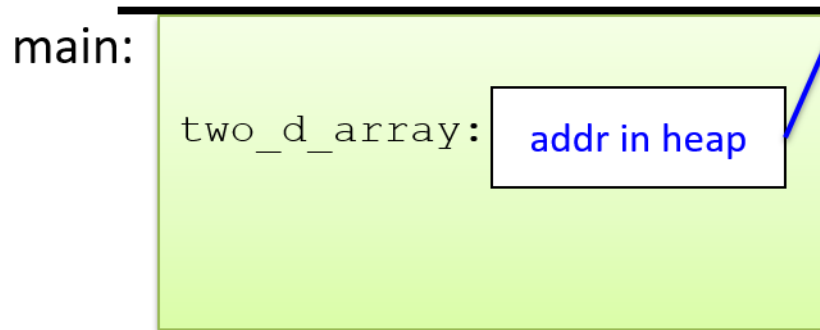
- If we want a dynamic array of ints:
 - declare `int *array = malloc(N * sizeof(int))`
- So... if we want an array of int pointers:
 - declare `int **array = malloc(...)`

Two-dimensional array alternative

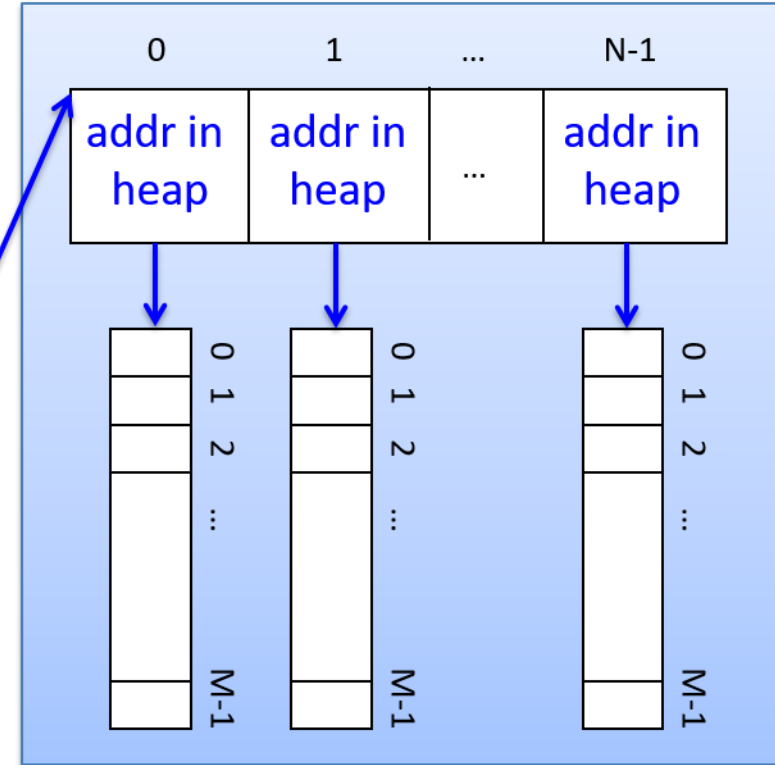
- If we want a dynamic array of ints:
 - declare `int *array = malloc(N * sizeof(int))`
- So... if we want an array of int pointers:
 - declare `int **array = malloc(N * sizeof(int *))`
 - The type of `array[0]`, `array[1]`, etc. is: `int *`
 - For each one of those, we can malloc an array of ints:
 - `array[0] = malloc(N * sizeof(int))`

Two-dimensional array alternative

```
int **two_d_array;  
  
two_d_array = malloc(sizeof(int *) * N);  
for (i=0; i < N; i++) {  
    two_d_array[i] = malloc(sizeof(int) * M);  
}
```



Stack



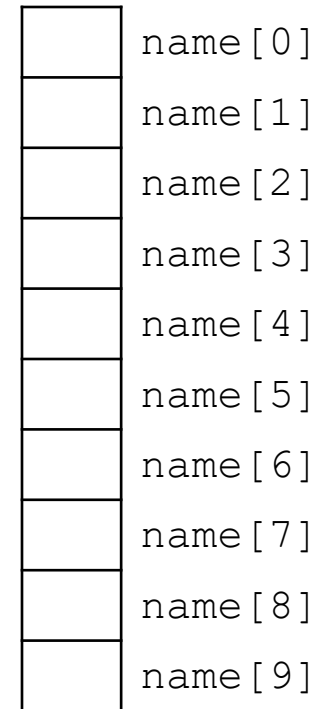
Heap

Two-dimensional arrays

- We'll use BOTH methods in future labs.

Strings

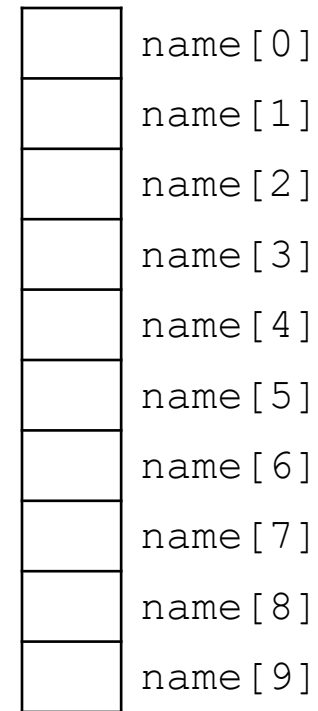
- Strings are *character arrays*
- Layout is the same as:
 - `char name[10];`
- Often accessed as `(char *)`



String Functions

- C library has many built-in functions that operate on char *'s:
 - strcpy, strdup, strlen, strcat, strcmp, strstr

```
char name[10];  
strcpy(name, "CS 31");
```



String Functions

- C library has many built-in functions that operate on char *'s:
 - strcpy, strdup, strlen, strcat, strcmp, strstr

```
char name[10];  
strcpy(name, "CS 31");
```

- Null terminator (\0) ends string.
 - We don't know/care what comes after

C	name[0]
S	name[1]
	name[2]
3	name[3]
1	name[4]
\0	name[5]
?	name[6]
?	name[7]
?	name[8]
?	name[9]

String Functions

- C library has many built-in functions that operate on char *'s:
 - strcpy, strdup, strlen, strcat, strcmp, strstr
- Seems simple on the surface.
 - That null terminator is tricky, strings error-prone.
 - Strings used everywhere!
- You will implement these functions in a future lab.

Up next...

- New topic: Storage and the Memory Hierarchy